行列・2次元格子上の数値データを C のプログラムでどう扱うべきか

桂田 祐史

2017年3月6日, 2017 年3 月6 日

1　はじめに — この文書の目的

数値計算のプログラミングをしているとき、行列と2次元格子上の数値データを扱う必要がしばしば生じる。素朴に考えると、2次元配列で扱うことが思い浮かぶが、色々な制限事項・注意事項がある。

説明する時間がないときは、次のような「私の結論 (お勧めの対応策)」を述べて、それに従ったサンプル・プログラムを示すことにしている。

私の結論 2017 年版

行列と2次元格子上の数値データを扱うために

- C ではポインターのポインターを使おう。
- C++ では、その (ポインターのポインター) 外に、適当なクラス・ライブラリィを使うという手もある。現時点での私のお勧めは **Eigen** である。

Eigen[1] については、別の解説を用意するとして、この文書では、C でなぜポインターのポインターを使うことを勧めるか、理由を述べる。

(桂田 [1] の中で少し書いてあるが、規格の移り変わりについて良く状況判断できなかったこともあり、今回少し時間をかけてまとめてみた。)

2　C 言語での1次元配列 (とそれに似たもの)

ベクトルのような、1次元に連続に並んでいるデータを C 言語で扱う場合に、言語のどのような機能が利用できるか、復習しておこう。

[1] http://eigen.tuxfamily.org/

例 (一部 ... で略しているので、このままコンパイルは出来ないけど)

```
/* 3 つのやり方の例示 */
#include <stdlib.h> // malloc() に必要

#define N (10)
double a[N]; // 静的変数

int main(void)
{
  ...
  double b[N]; // 自動変数
  double *c;   // c は double へのポインター
  c = malloc(sizeof(double) * N); // malloc() でメモリを割り当てる。
  if (c == NULL) // 割り当てに成功したかどうかチェックできる。
  ...
  for (i = 0; i < N; i++)
    c[i] = a[i] + b[i]; // a, b, c どれも同じように使える。
  ...
  free(c); // 不要になったらメモリを解放する必要がある。
}
```

$0 \leq i \leq$ N-1 を満たす i に対して、a[i], b[i], c[i] が使える。そういう意味では似ているけれど、違いがある。

- a のサイズは割と大きく出来る。しかしサイズは (コンパイル時に決まるように) 定数である必要がある。
- b のサイズはあまり大きくできない (メモリのスタック領域に配置されるため)。C99 以前の C の場合、サイズは定数でないといけない。C99 で可変長配列が導入されたので、サイズが定数である必要はなくなったが、C11 で可変長配列の機能はオプションに戻されたので、注意が必要である。
- c はポインター変数である、厳密には配列変数でないが、配列変数とほぼ同じように使える。c のサイズはかなり大きく取れる (メモリのヒープ領域に配置されるため)。サイズが定数である必要はない。

実際上はどのやり方を採用してもあまり問題にならないことが多い、と思われる。

3 C 言語での2次元配列 — 悩ましさの解説

それでは行列と 2 次元格子上の数値データはどうすれば良いだろうか。素朴に考えると 2 次元配列で扱えば良さそうだが…

3.1 サイズが事前に分かっている場合

サイズが事前に分かっていれば (定数であるならば)、

```
#define M (10)
#define N (10)

double a[M][N]; // 静的変数

int main()
...
```

のようにするのが簡単である。

でも残念ながらこういう場合はあまり多くない。数値実験をする場合、行列にしても 2 次元格子上の数値データにしても、サイズを色々と変えて実験すべきである (この点は肝に命じて欲しい)。そういうときに、毎回 M, N の値を書き換えてコンパイルし直す、というのは拙い。

3.2 大きく確保して、その一部を使う戦略

サイズが事前に分かっていなくても、それがそんなに大きくなければ

```
#define MAXM (10)
#define MAXN (10)

int main(void)
{
   double b[MAXM][MAXN]; // これは 800 バイトのメモリを占める（と分かる？）
   int m, n, i, j;
   printf("m,n="); scanf("%d%d", &m, &n);
   if (m > MAXM || n > MAXN) { // 一応エラー・チェック
     ...
   }
   for (i = 0; i < m; i++)
     for (j = 0; j < n; j++)
        ...
```

のようにすることが考えられる。つまり、記憶領域を大きめに確保して、その一部だけを使う、ということである。

でもあまりお勧めできない。ベクトルならば大したことにはならないだろうけれど、行列の場合はサイズの制限は案外きつい。自分が現在使っている C 言語処理系で、スタック領域のデフォールトのサイズがいくらであるか、即答できる人は多くないだろう (僕は覚えていません — 今使っている Mac で調べたら 8MB だった)。

サイズが事前にわからない、ある程度以上大きい、という場合、次のようにする方がよい。

```
#define MAXM (200)
#define MAXN (200)
double a[MAXM][MAXN]; // 静的に大きめに確保

int main(void)
{
   int m, n, i, j;
   printf("m,n="); scanf("%d%d", &m, &n);
   ...
```

上と変わらないように感じるかもしれないが、a を関数の外で定義する (静的変数として定義する) ことが重要なポイントである。これならばスタック領域のサイズは気にならない。

このやり方はまあまあ使える。しかし関数に引数として渡すときに少し悩ましいことが起こる。これは案外、説明に手間取るので、項を改める。

3.3 整合配列、可変長配列

例えば2つの n 次正方行列 a, b の積を計算して、その結果を c に代入する関数を考えてみよう。Fortran の場合は次のようにすれば良い。

サブルーチン

```
      subroutine mulmat(c, a, b, ldim, n)
      integer ldim, n
      real*8 c(ldim,*), a(ldim,*), b(ldim,*)
      integer i,j,k
      real*8 s
      do i=1,n
        do j=1,n
          s=0.0
          do k=1,n
            s=s+a(i,k)*b(k,j)
          end do
          c(i,j)=s
        end do
      end do
```

サブルーチンを呼び出す側 (例えばメイン・プログラム)

```
      call mulmat(c, a, b, MAXM, n)
```

行列としての次数 n 以外に、2次元配列の1番目の次元 (the leading dimension と呼ばれる — だから ldim) MAXM をサブルーチンに渡す必要がある。c(ldim,*) のように、配列のサイズを引数 ldim を使って指定出来るのが「整合配列」と呼ばれる機能である。

さて、それで C の場合であるが、C には Fortran のような整合配列の機能はない。しかし C99 で導入された可変長配列の機能を使うとほぼ同じことが出来る。

関数

```
void mulmat(int n, int cols,
            double a[][cols], double b[][cols], double c[][cols])
{
  int i, j, k;
  double s;
  // 略 for (i=0; i<n; i++) for (j=0; j<n; j++) { s=0.0; for (k=0;...
}
```

関数を呼び出す側 (例えば main() から)

```
  mulmat(n, MAXN, a, b, c);
```

2 次元配列の要素がメモリの中でどういう順番で配置されるか、Fortran と C で違いがあるため (Fortran は column major, C は row major — この辺も解説を書くべきか)、サブルーチン・関数に渡すべき情報が、a[MAXM][MAXN] のうちの MAXM, MAXN のどちらであるかに違いが出ている。

古い C では、可変長配列が使えなかったため、個人的には 4 節で紹介するやり方を使い (M 師匠に教わったんだっけ)、学生にも勧めていた。そのうちに GCC で可変長配列が導入され、C99 で規格化されたので、この項で説明したやり方が可能になったが、C99 の次の規格 C11 では、可変長配列の機能はオプショナルとされたので、このやり方を人に勧めて良いか、迷うところである (使っている人を見て、止めようとまでは思わない)。

念のため: 可変長配列の次のような利用

上では、整合配列の真似をするために可変長配列機能を用いたが、次のように利用することも考えられる。

```
void nanika(int n, double a[][n])
{
  ...
}

int main(void)
{
  int m, n;
  printf("m,n="); scanf("%d%d", &m, &n);
  double a[m][n];

  nanika(n, a);
```

細かい説明は省略するが、次の特徴を持つことは理解できるであろう。

- 可変長配列を使っているので、C99 でないと安心して使えない。
- スタック領域におかれるので、あまり大きなサイズの配列は使えない。

(今回の問題の解決には役立たない。)

3.4 脱線のような大事な話のような: GLSC 利用上の問題

現象数理学科ローカルの話になるけれど、上のやり方で 2 次元配列を用意して、GLSC の等高線・鳥瞰図描画でハマっている人が何人かいた。

```
g_contln(-1.0, 1.0, -1.0, 1.0, u,
         N_x + 1, N_y + 1, 0.32 * i - 1.12);
```

とか

```
g_hidden(1.0, 1.0, 0.4, -1.0, 1.0, 5.0, 25.0, 20.0, 20.0, 20.0,
         150.0, 100.0, u, N_x + 1, N_y + 1,
         1, G_SIDE_NONE, 2, 1);
```

とかしたとき、GLSC 側は、u では、2 次元格子上の数値データが u の示す場所に連続的に並んでいる、として処理を行うので、

```
double u[N_x+1][N_y+1];
```

として宣言してあれば良いのだけど、

```
double u[MAXN_x+1][MAXN_y+1]; // 大きめに確保して、一部を使うつもり
```

と宣言した場合は (MAXN_x=N_x, MAXN_y=N_y でない限り) 正しく動かないのである。

…実は GLSC を作った人達は、double u[MAXN_x+1][MAXN_y+1]; とした場合にも対応できるようにプログラムを書いてあるが、それは裏の方に隠されていて、通常は使えないようになっている (static な関数になっている)。

4 ポインターのポインターを使おう

さて、それでどうするか、という最初の問題に戻るけれど、C の場合はポインターのポインターを使うことを勧める。

```
#include <stdlib.h> // malloc() に必要

int main(void)
{
  int m,n, i;
  double **a;     // double へのポインターのポインター
  double *a_data; // double へのポインター
  printf("m,n="); scanf("%d%d", &m, &n);
  a = malloc(sizeof(double *) * m); // double へのポインター m 個分の領域確保
  a_data = malloc(sizeof(double) * (m * n)); // double m*n 個分の領域確保
  for (i = 0; i < m; i++)
    a[i] = a_data + (i * n);
  // これで a[i][j] (0≦ i≦ m-1, 0≦ j≦ n-1) が使える
```

すごく面倒に感じるかもしれないが、こういうのは関数にしてしまえば良い。次のコードでは、new_matrix() という関数を作って、その使い方を示している。

```
#include <stdlib.h> // malloc() のため

typedef double *vector;
typedef vector *matrix; // matrix って行列という意味です。

// m行n列の行列（あるいは2次元格子上の数値データ）を記憶する領域の割り当て
matrix new_matrix(int m, int n)
{
  matrix a;
  vector a_data;
  int i;
  a = malloc(sizeof(double *) * m);
  if (a == NULL) // せっかくなのでエラー・チェックも
    return NULL;
  a_data = malloc(sizeof(double) * (m * n)); // 必要十分なサイズ
  if (a_data == NULL) {
    free(a);
    return NULL;
  }
  for (i = 0; i < m; i++)
    a[i] = a_data + (i * n);
  return a;
}

void delete_matrix(matrix a)
{
  free(a[0]); free(a); // a[0] は a_data であるから
}

int main(void)
{
  int m,n;
  matrix a, b, c;
  printf("m,n="); scanf("%d%d", &m, &n);
  a = new_matrix(m, n); // 関数にしておけば3つ用意するのも簡単
  b = new_matrix(m, n);
  c = new_matrix(m, n);
  if (a == NULL || b == NULL || c == NULL) {
    ...
  }
  ...
  free_matrix(a); free_matrix(b); free_matrix(c); // 要らなくなったら解放
```

(int main(void) の前はコピペして使えば良い。)
このやり方の特徴をまとめておく。

(1) 可変長配列の機能は使っていないので、C のどの規格でも動く。
(ポインターは、古くから C に備わっている機能で、決してなくならないだろう。)

(2) ポインター m + 1 個だけメモリーが (追加で) 必要になる[2]。— 大したことではない。

(3) 関数を自作する場合、引数のやり取りは簡単である。

```
void mulmat(int n, matrix a, matrix b, matrix c)
{
  int i, j, k;
  double s;
  // 略 for (i=0; i<n; i++) for (j=0; j<n; j++) { s=0.0; for (k=0;...
}
```

自作でない関数を使う場合にも、少しの工夫で対応できる場合が多い。例えば、上で話題に出した、GLSC の g_contln() や g_hidden() を使う場合、2 次元格子上の数値データが

```
u = new_matrix(nx+1, ny+1);
```

として確保された u という matrix に記憶されているとして

```
g_hidden(1.0, 1.0, 0.4, -1.0, 1.0, 5.0, 25.0, 20.0, 20.0, 20.0,
         150.0, 100.0, u[0], nx + 1, ny + 1,
         1, G_SIDE_NONE, 2, 1);
```

のように (配列を渡すべきところで) u[0] を渡せば良い (new_matrix() の中の a_data が渡されることになる)。

A C++の場合

C++の場合は、色々便利な線型演算用のクラス・ライブラリィがあるので、そういうものを使えば良いと思うが、C++ は基本的に C の上位互換なので、上で説明しているやり方は C++ でも使える。

ただし C++ では、メモリの割り当てと解放は (C のように malloc(), free() を使うのでなく)、new と delete を使うのが普通なので、それらを使うように書き直しておく。

(NULL でなく、nullptr とすべきなのかな？)

[2]大きく確保して、その一部だけ使う戦略に比べれば無視可能である。これまで個人的に使ったことのある一番大きな行列は、250 万次の正方行列であった。その場合、ポインターに 20MB 近くのメモリーが使われるが、これくらいは大したことないであろう (そのとき、行列の要素の記憶に数十 GB 使っていたので、それに比べれば)。

C++ 版 new_matrix()

```
#include <iostream>
using namespace std;

typedef double *vector;
typedef vector *matrix; // matrix って行列という意味です。

matrix new_matrix(int m, int n)
{
  matrix a = new vector [m];
  if (a == NULL) {
    return NULL;
  }
  vector ap = new double [m*n];
  if (ap == NULL) {
    delete [] a;
    return NULL;
  }
  for (int i = 0; i < m; i++)
    a[i] = ap + i * n;
  return a;
}

void delete_matrix(matrix a)
{
  delete [] a[0];
  delete [] a;
}
```

参考文献

[1] 桂田祐史：C 言語これくらいは覚えよう, http://nalab.mind.meiji.ac.jp/~mk/labo/text/cminimum/ (2005 年〜).